Sascha Grunert

1.2K Followers About Follow

Demystifying Containers — Part III: Container Images

Sascha Grunert Sep 17, 2019 · 17 min read

This series of blog posts and corresponding talks aims to provide you with a pragmatic view on containers from a historic perspective. Together we will discover modern cloud architectures layer by layer, which means we will start at the Linux Kernel level and end up at writing our own secure cloud native applications.

Simple examples paired with the historic background will guide you from the beginning with a minimal Linux environment up to crafting secure containers, which fit perfectly into todays' and futures' orchestration world. In the end it should be much easier to understand how features within the Linux kernel, container tools, runtimes, software

> *Every related resource can be found in the corresponding GitHub repository.*

Part III: Container Images

This third blog post (and talk) will be all about container images. As usual, we start with the historic background and the evolution of different container image formats. Afterwards, we will check out what is inside of the latest Open Container Initiative (OCI) image specification by crafting, modifying and pulling apart our self-built container image examples. Besides that, we will learn some important best practices in modern container image creation by utilizing tools like buildah, podman and skopeo.

Introduction

The *Cloud Native* world is following multiple trends these days, whereas topics like “running Continuous Integration (CI) and Deployment (CD) pipelines inside of Kubernetes” gain more and more attention. There is an increasing popularity to unify testing and deployment infrastructure, which finds its motivation in a lot of rapidly developing open source projects which try to achieve this goal. As the consumers of those software projects, companies have to create and maintain an efficient software life cycle, from developing their own source code to deploying cloud based applications which should be used by probably millions of people worldwide. There are multiple issues which would come to our mind if we think about this scenario, like “How to make our systems secure?” and “Which infrastructure solution should I use to make it secure?”. To provide an answer to these questions we first have to take one step back:

> “As a company, am I able to afford the costs of a healthy Dev(sec)ops strategy?”

Probably not?

> “Do my developers really understand what they do if they set up a fully automated CD pipeline, which builds, pushes and canary deploys the latest microservice artifacts in one rush directly to the production environment?”

As of today, getting rapid access to working infrastructure is not the same problem any more like ten years ago. Solutions like Amazon Web Services and the Google Cloud offer a practical and well maintained solution for mostly every use case, which can be used safely within a few clicks. Nevertheless, those kinds of solutions are mostly not open source which always imply some feeling of non-transparency.

Beside selecting an infrastructure solution, the hardest part might be finding a good way in developing and deploying software efficiently and securely. The security aspect is sometimes highly underestimated, since people might think that CI/CD pipelines do not need to have the same security constraints like setups built for production use cases. That might be true for some specially secured on-premise environments, but in the world of Kubernetes we want to move our application easily without considering proprietary infrastructure related limitations. To achieve this, we have to apply the same security patterns for every piece of the overall deployment independently if we provide an application on a test cluster, production environment, or even developing the CI/CD software itself.

All these applications needs to be deployed in container images, since mainly everything at scale runs in containers today. Now, there might be questions popping up like: "How should my container based deployment look like?", "Which base distribution is the best for my application?" or "How are these security issues related to containers and images?". To answer these questions, we will start from the beginning and afterwards dive into the bright world of container images by simply trying it out ourselves!

A Brief History

As we already know from the past blog post, the basic concept of containers has been buzzing around for a while when a tool called Docker was invented. Nevertheless, they were the first back in 2013 who were able to pack containers into images. This enabled users to move container images between machines, which marked the birth of container based application deployments. As of today, multiple versions of the container image format exist, whereas the Docker developers decided to create version 2 (V2) schema 1 of the image manifest back in 2016 and therefore deprecate version 1. After multiple iterations of various image format improvement approaches, schema 2 has been released to supersede the existing schema 1 in 2017. This second schema version had model where the images' configuration can be hashed to generate an ID for the image. The V2 image specification was later donated to the Open Container Initiative (OCI) as base for the creation of the OCI image specification. 2017 was the year where the OCI image format v1.0.0 has been released, so let's take a look what it's all about.

The OCI Image Specification

Generally, OCI container images consist of a so called *manifest*, a *configuration* a set of *layers* and an optional *image index*. That's all we need to describe a container image, whereas everything except the layers are written in JavaScript Object Notation (JSON).

During the image creation process, all the parts of the specification will be bundled together into a single artifact, the *container image*. After that, the bundle can be downloaded from a network resource like a container registry or be baked into a tarball to pass it around.

Okay, let's examine the contents of an image by utilizing the awesome container image tool skopeo:

```
> skopeo copy docker://saschagrunert/mysterious-image oci:mysterious-
image
Getting image source signatures
Copying blob
sha256:0503825856099e6adb39c8297af09547f69684b7016b7f3680ed801aa310ba
aa
 2.66 MB / 2.66 MB
[=========================================================] 0s
Copying blob
sha256:6d8c9f2df98ba6c290b652ac57151eab8bcd6fb7574902fbd16ad9e2912a67
53
```

```
25
 850 B / 850 B
[==============================================================] 0s
Writing manifest to image destination
Storing signatures
```

In the output we can see that the mentioned manifest and the configuration got pulled separately from some other *blobs* out of the registry. This gives us an indicator about how the images are stored remotely. But what did we download at all?

```
> tree mysterious-image
mysterious-image
├── blobs
│   └── sha256
│       ├──
0503825856099e6adb39c8297af09547f69684b7016b7f3680ed801aa310baaa
│       ├──
6d8c9f2df98ba6c290b652ac57151eab8bcd6fb7574902fbd16ad9e2912a6753
│       ├──
703c711516a88a4ec99d742ce59e1a2f0802a51dac916975ab6f2b60212cd713
│       └──
8b2633baa7e149fe92e6d74b95e33d61b53ba129021ba251ac0c7ab7eafea825
├── index.json
└── oci-layout
```

Every *blob* is located in a single `sha256` directory as flat hierarchy, whereas an image index (`index.json`) and `oci-layout` coexist separately. The latter can be ignored for now since it only contains the layout version of the image: `{"imageLayoutVersion": "1.0.0"}` .

Now, we need to find out how the image looks from the inside. But how to do that? At first, let's check out what the `index.json` of the downloaded image contains:

```
> jq . mysterious-image/index.json

{
  "schemaVersion": 2,
  "manifests": [
    {
      "size": 502,
      "platform": {
        "architecture": "amd64",
        "os": "linux"
      }
    }
  ]
}
```

It looks like that the image index is just a higher-level manifest, which contains pointers to more specific image manifests. These manifests seem to be valid only for their dedicated target platforms, which contain a specific operating system (`os`) like `linux` and an `architecture` , like `amd64` . As already mentioned, the *image index* is fully optional. The `mediaType` can be seen as identifier of the object, whereas the digest is the unique reference to it.

When it comes to multi-architecture images in terms of docker, then it works mainly the same as for OCI images, except that their media type is `application/vnd.docker.distribution.manifest.list.v2+json` , which contains all the manifests for the different target platforms. During an image pull, docker automatically selects the correct manifest based on the host architecture, which makes multi-architecture images work like a charm.

Let's have a look at the image manifest which is linked in the index:

```
> jq . mysterious-
image/blobs/sha256/703c711516a88a4ec99d742ce59e1a2f0802a51dac916975ab
6f2b60212cd713

{
  "schemaVersion": 2,
  "config": {
    "mediaType": "application/vnd.oci.image.config.v1+json",
    "digest":
"sha256:8b2633baa7e149fe92e6d74b95e33d61b53ba129021ba251ac0c7ab7eafea
825",
    "size": 850
  },
        "digest":
"sha256:0503825856099e6adb39c8297af09547f69684b7016b7f3680ed801aa310b
aaa",
      "size": 2789742
    },
    {
      "mediaType": "application/vnd.oci.image.layer.v1.tar+gzip",
      "digest":
"sha256:6d8c9f2df98ba6c290b652ac57151eab8bcd6fb7574902fbd16ad9e2912a6
753",
      "size": 120
    }
  ]
}
```

The image *manifest* provides the location to the *configuration* and set of *layers* for a single container image for a specific architecture and operating system. The `size` field, as the name stated, indicates the overall size of the object.

This means that we're now able to obtain further information about the image *configuration*, like this:

```
> jq . mysterious-
image/blobs/sha256/8b2633baa7e149fe92e6d74b95e33d61b53ba129021ba251ac
0c7ab7eafea825

{
  "created": "2019-08-09T12:09:13.129872299Z",
  "architecture": "amd64",
  "os": "linux",
  "config": {
    "Env": [

"PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
    ],
    "Cmd": ["/bin/sh"]
  },
  "rootfs": {
    "type": "layers",
    "diff_ids": [

"sha256:1bfeebd65323b8ddf5bd6a51cc7097b72788bc982e9ab3280d53d3c613adf
fa7",
  },
  "history": [
    {
      "created": "2019-07-11T22:20:52.139709355Z",
      "created_by": "/bin/sh -c #(nop) ADD
file:0eb5ea35741d23fe39cbac245b3a5d84856ed6384f4ff07d496369ee6d960bad
in / "
    },
    {
      "created": "2019-07-11T22:20:52.375286404Z",
      "created_by": "/bin/sh -c #(nop)  CMD [\"/bin/sh\"]",
      "empty_layer": true
    },
    {
      "created": "2019-08-09T14:09:12.848554218+02:00",
      "created_by": "/bin/sh -c echo Hello",
      "empty_layer": true
    },
    {
      "created": "2019-08-09T12:09:13.129872299Z",
      "created_by": "/bin/sh -c touch my-file"
    }
  ]
}
```

On the top of the JSON we find some metadata, like the `created` date, the `architecture` and the `os` of the image. The environment (`Env`) as well as the command (`Cmd`) to be executed can be found in the `config` section of the configuration. This section can contain even more parameters, like the working directory (`WorkingDir`), the `User` which should run the container process or the `Entrypoint` . This means, if you create an container image via a `Dockerfile` , then all these parameters will be converted into the JSON based *configuration*.

We can also see the `history` executed during image creation since OCI images are just an ordered collection of root filesystem changes and the corresponding execution parameters. The `history` indicates that we added a file to the base image as well as executed an `echo` and `touch` command in order. The `empty_layer` boolean is used to mark if the history item created a filesystem diff. The configuration also indicates that the `rootfs` is split into `layers` , whereas the `diff_ids` are different from the actual layer digests, because they reference the digests of the uncompressed `tar` archives.

the layers in parallel, but they have to be extracted sequentially. This takes time and is a significant drawback when it comes to pulling images from remote locations.

Now, let's have a look at the first layer, where we would expect the root filesystem (`rootfs`):

```
> mkdir rootfs
> tar -C rootfs -xf mysterious-
image/blobs/sha256/0503825856099e6adb39c8297af09547f69684b7016b7f3680
ed801aa310baaa
> tree -L 1 rootfs/
rootfs/
├── bin
├── dev
├── etc
├── home
├── lib
├── media
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin
├── srv
├── sys
├── tmp
├── usr
└── var
```

Looks like a real file system layout, right? And it truly is, so we can easily examine the base distribution of the container image:

```
> cat rootfs/etc/os-release
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.10.1
PRETTY_NAME="Alpine Linux v3.10"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
```

```
> mkdir layer
> tar -C layer -xf mysterious-
image/blobs/sha256/6d8c9f2df98ba6c290b652ac57151eab8bcd6fb7574902fbd1
6ad9e2912a6753
> tree layer
layer
└── my-file
```

The `layer` contains only the file which which has been added on running `touch my-file`, like we can see it in the `history` section of the image configuration.

Now, we have everything at hand to reverse engineer the `Dockerfile` from which the image has been built. In the end, the recovered `Dockerfile` could look like this:

```
FROM alpine:latest
RUN echo Hello
RUN touch my-file
```

But is this really everything? In our case yes, but at the end of the day it depends on the tool which builds the container image, since Docker is not the only one which is capable of doing that.

One rising star of the container image builders is buildah, which nicely follows the Unix Philosophy and concentrates on the single dedicated task of building OCI container images.

Buildah

flexible solutions when it comes to building container images these days. Companies like SUSE decided to contribute to buildah as well to integrate it into their container ecosystem and provide a well maintained alternative to Docker based image builds.

So, if we're migrating from the Docker world, then it is worth to mention that it is surely possible to build container images using standard Dockerfiles with buildah. Let's give that a try:

```
> cat Dockerfile

FROM alpine:latest
RUN echo Hello
RUN touch my-file
```

This is how our Dockerfile looks like, whereas we can build it by executing `buildah bud` in the same directory:

```
> buildah bud
STEP 1: FROM alpine:latest
Getting image source signatures
Copying blob 050382585609 done
Copying config b7b28af77f done
Writing manifest to image destination
Storing signatures
STEP 2: RUN echo Hello
Hello
STEP 3: RUN touch my-file
STEP 4: COMMIT
Getting image source signatures
Copying blob 1bfeebd65323 skipped: already exists
Copying blob dcf360a74dad done
Copying config 534e2776ab done
Writing manifest to image destination
Storing signatures
534e2776ab1e7787f4682e9195fd2ff996a912fdb93abe301d8da6bdf0d32c92
```

This looks pretty familiar when comparing to a `docker build` output, right?

running containers with `buildah ps`.

Running containers versus building them

Wait, listing running containers? Wasn't buildah meant to be a container *building* and not a container *running* tool? Well, that's what it's all about: If we build container images, either with docker or buildah, we actually spawn intermediate containers which will be modified during their runtime. Every modification step can now create new history entries for the image, whereas new layers are created by filesystem modifications.

This can be a security issue, because in the world of the Docker daemon it would be possible to inject anything in a currently building container by just hijacking the build process. Imagine we have a Dockerfile which contains operations which takes a larger amount of time, like installing `clang` on the latest version of Debian:

```
FROM debian:buster
RUN apt-get update -y && \
    apt-get install -y clang

> docker build -t clang .
[…]
```

During the installation process, we can see that there is a docker container running:

```
> docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS                    NAMES
05f4aa4aa95c        54da47293a0b        "/bin/sh -c 'apt-get…"   11
seconds ago      Up 11 seconds
interesting_heyrovsky
```

Okay, then let' jump into that container:

We can do any container modification right in the currently building container. After the successful built of the docker container image, we can verify that the created `my-file` is there:

```
> docker run clang cat my-file
Hello world
```

Oh my, a mean side-effect of that kind of hijacking is that the `echo` command we executed is not part of the containers command history, but of the layer created during the `apt-get` command invocations.

Building containers without Dockerfiles

Buildah provides dedicated subcommands for mainly every available command we can execute within a Dockerfile, like `RUN` or `COPY`. This opens up an immense amount of flexibility, because it is now possible to overcome this obstacle of single, huge, blowing-up Dockerfiles and split-up the build process in nicely looking pieces. We can use every UNIX tool we can imagine in between of the different container image build steps, to manipulate the target containers filesystem or run tasks in their native environments and consume their outputs for additional container build steps.

Let's give it a try by creating a new base container on top of Alpine Linux:

```
> buildah from alpine:latest
alpine-working-container
```

We now have a running working container called `alpine-working-container`, which we can list with `buildah ps`:

```
bc50127e88aa     *     b7b28af77ile docker.io/library/alpine:latest
alpine-working-container
```

Now, we can run commands within that container, for example this one to demonstrate that it is actually an Alpine Linux distribution:

```
> buildah run alpine-working-container cat /etc/os-release
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.10.1
PRETTY_NAME="Alpine Linux v3.10"
HOME_URL="https://alpinelinux.org/"
BUG_REPORT_URL="https://bugs.alpinelinux.org/"
```

Or create a new file within the container:

```
> echo test > file
> buildah copy alpine-working-container file /test-file
86f68033be6f25f54127091bb410f2e65437c806e7950e864056d5c272893edb
```

By default, buildah does not create new history entries to the container, which means that the order of the commands or how often they are called have no influence on the layers of the resulting image. It is worth to mention that this behavior can be changed by adding the `--add-history` command line flag or setting the `BUILDAH_HISTORY=true` and environment variable.

But generally everything seems to work, so then let's `commit` our new container image to finalize the build process:

```
> buildah commit alpine-working-container my-image
Getting image source signatures
Copying blob 1bfeebd65323 skipped: already exists
Copying blob 78dc6589d67a done
Copying config a81c56b613 done
Writing manifest to image destination
```

Now, the new container image `my-image` should be available within our local registry:

```
> buildah images
REPOSITORY                  TAG      IMAGE ID       CREATED
SIZE
localhost/my-image          latest   a81c56b613a9   36 seconds ago
5.85 MB
docker.io/library/alpine    latest   b7b28af77ffe   4 weeks ago
5.85 MB
```

Buildah is able to push the container image into a Docker registry or into an local on-disk OCI format, like this:

```
> buildah push my-image oci:my-image
Getting image source signatures
Copying blob 1bfeebd65323 done
Copying blob 78dc6589d67a done
Copying config a81c56b613 done
Writing manifest to image destination
Storing signatures
```

The created directory `my-image` contains now a fully OCI compatible image layout, with *image index*, *configuration* a set of *layers* and a *manifest*.

```
> jq . my-image/index.json
{
  "schemaVersion": 2,
  "manifests": [
    {
      "mediaType": "application/vnd.oci.image.manifest.v1+json",
      "digest":
"sha256:05110b1d784a242a5608590ff5c6c8ed6cf8f08f3c5692a9b54bcc514204a
467",
      "size": 493,
      "platform": {
        "architecture": "amd64",
        "os": "linux"
      }
```

Buildah wouldn't be buildah if we could not pull the image back from such a directory structure. To demonstrate this, let's first remove the image `my-image` from buildah's local registry:

```
> buildah rmi my-image
[…]
```

And then pull it in again:

```
> buildah pull oci:my-image
[…]

> buildah images
REPOSITORY                    TAG      IMAGE ID       CREATED
SIZE
docker.io/library/my-image    latest   a81c56b613a9   9 minutes ago
5.85 MB
```

That we `commit` the container does not mean that the `alpine-working-container` is not available any more, we can still modify it and commit the intermediate state later on. This is great for building multi-staged container images, where the build results of one image can be re-used by another image. It also streamlines the build processes for containers, where a simple bash script has much more power and flexibility than a single Dockerfile.

Let's check if our working container is still running, by interactively spawning a shell within it:

```
> buildah run -t alpine-working-container sh
/ # ls
bin        etc        lib        mnt        proc       run        srv
test-file  usr        dev        home       media      opt
root
```

Nice, our modifications are still available!

With buildah, it is even possible to mount the containers filesystem locally to get rid of the build context limitation of a Docker daemon. So, we're able to do something like this:

```
> buildah unshare --mount MOUNT=alpine-working-container
> echo it-works > "$MOUNT"/test-from-mount
> buildah commit alpine-working-container my-new-image
```

First of all, we had to call `buildah unshare`. What this actually does is creating a new user namespace, which allows us to mount the filesystem as current non-root user. The `--mount` automatically does the mounting for us, whereas we expose the resulting path to the `$MOUNT` environment variable.

After that, we're free to modify the filesystem locally as we want it. Once we're done, then the changes need to be committed, whereas the mount automatically gets removed when leaving the `buildah unshare` session.

That's it, we successfully modified the containers filesystem via a local mount! We can verify this via testing for the `test-from-mount` file we created within the working container.

```
> buildah run -t alpine-working-container cat test-from-mount
it-works
```

A large benefit from using this approach is that we don't have to copy-around all the data, for example when sending the build context to the Docker daemon. This can take quite some time on slow hardware and doesn't provide any benefit if the daemon runs on the local machine.

buildah'ception

Buildah does not have any daemon at all, which reduces the security-related attack surface by minimizing the projects' overall complexity. Having no daemon at all does also mean that we don't have to mount the `docker.sock` into a container to do something with the Docker's command line interface (CLI). It's easy to simply use buildah and avoid such anti-patterns at all to keep a stable security foundation, especially in projects which are crossing the border between CI/CD and production environments.

Let's do it, we can run buildah within a container running on top of buildah! To do this, let's first install buildah in a container, which is already running by buildah:

```
> buildah from opensuse/tumbleweed
tumbleweed-working-container

> buildah run -t tumbleweed-working-container bash
# zypper in -y buildah
```

Now, buildah should be ready to be used within that container. Please note that we have to choose the storage driver `vfs` within the container to have a working filesystem stack:

```
# buildah --storage-driver=vfs from alpine:latest
alpine-working-container
# buildah --storage-driver=vfs commit alpine-working-container my-
image
[…]
```

Wow! We built our first container image inside a container image, without any additional tricks!

storage to get the image on our local machine.

First, let’s push the image into the container under `/my-image`:

```
# buildah --storage-driver=vfs push my-image oci:my-image
[…]
```

Then we exit the containerception and copy the built image out of the running working container by mounting its filesystem:

```
> buildah unshare
> export MOUNT=$(buildah mount tumbleweed-working-container)
> cp -R $MOUNT/my-image .
> buildah unmount tumbleweed-working-container
```

We can now pull the image from the directory directly into buildah’s local registry:

```
> buildah pull oci:my-image
[…]
> buildah images my-image
REPOSITORY                   TAG      IMAGE ID       CREATED
SIZE
docker.io/library/my-image   latest   e99b6777e420   7 minutes ago
5.85 MB
```

We finally achieved to built a container image within a running container!

Another great feature of buildah is something we did not mention at all until now: During the whole demonstration, there was nothing like a `sudo` in one of our `buildah` invocations. We run the tool completely in rootless mode, which means that everything necessary was done with the current users‘ set of permissions.

- `~/.config/containers` , for configuration
- `~/.local/share/containers` , for the storage

This is an incredible security enhancement, makes user dependent configuration possible and hopefully will be the common standard in the next couple of years.

Buildah has some more features we did not cover yet, like a nice Dockerfile CPP preprocessor macro support to de-compose a single Dockerfile into multiple ones. Feel free to try it out yourself or reach out to me if you have any further questions.

podman as buildah's interface

Podman targets to be a drop-in replacement to Docker. We don't cover it in detail here because this would consume too much of your valuable reading time. Nevertheless, podman uses buildah as API to provide native Dockerfile building support with `podman build` . This means that it shares the same storage under the hood to run the containers built with buildah. This is pretty neat, so we can actually run our previously built containers like this:

```
> podman run -it my-image sh
/ # ls
bin    dev    etc    home   lib    media  mnt    opt    proc   root
run
sbin   srv    sys    tmp    usr    var
```

It is worth to mention again that this runs in rootless mode too, which is pretty neat!

Conclusion

And that's it for the third part of the demystification of containers. We discovered the brief history of container images and had the chance to build our own containers using buildah. We should now have a good basic understanding what container images consist of. This is necessary to dive into further topics like container security and understand why Kubernetes features work like they do. For sure we did not have the chance to talk

I really hope you enjoyed the read and will continue following my journey into future parts of this series. Feel free to drop me a line anywhere you can find me on the internet. Stay tuned!

Docker Buildah Containers Kubernetes Oci